Lecture Set #9: Arrays Intro

This lecture set:
- Intro to arrays
- Copying arrays and making arrays bigger
- Array lengths and out-of-bounds indexing
- Passing arrays and array elements to a function
- Privacy Leaks
- Different levels of copy

Data Structures and Arrays

- **Data structures**: mechanisms for storing data in a structured way
- We have seen simple data structures implemented as classes:
  - `Rational.java`
    - Rational number data stored as numerator / denominator pair
  - **Arrays**: a very useful data structure provided by Java and other programming languages
    - Array: sequence of variables of the same type
      - homogeneous data structure
      - size (quantity) fixed when space is allocated
      - ordered
    - Individual elements of sequence can be referenced/updated/etc.
    - Arrays are objects (hence allocated on heap) with a reference on the stack
    - Like other objects, “instance variables” of array = cells in array are assigned default values (0 / null / etc.) when array created
Array Indexing

- Java provides a special syntax for uniformly accessing cells in an array
  - Declaration of a:
    ```java
    int[] a;
    ```
  - Allocation of space for array named a:
    ```java
    a = new int[5];
    ```
    - This creates five int variables "named": `a[0], a[1], a[2], a[3], a[4]`
  - To modify contents of cell #2 to 6 and cell #1 to 74:
    ```java
    a[2] = 6;
    a[1] = 74;
    ```
  - To use the contents of cell #2 and cell #1:
    ```java
    System.out.println("value = " + (a[1]-a[2]));
    ```
  - This access mechanism to the individual elements is called **array indexing**
    - In Java / C / C++, array cells are indexed beginning at 0 and going up to n-1 (n is number of cells)
    - Beware: start at 0! and end at one less than the size!!

Square Brackets: [ ] and length

- Three uses in Java:
  - Array variable declaration
    ```java
    int[] a;
    ```
  - Array object creation
    ```java
    new int[10];
    ```
  - Array indexing
    ```java
    a[0]
    ```
    - array also has `a.length` holds the amount of space currently allocated for that array
Alternate Declaration Syntax

- To maintain consistency with C / C++, following declaration of array variables also possible
  
  ```
  int grade[];
  ```
  
  Compare to Java standard:
  
  ```
  int[] grade;
  ```

- Java standard generally preferred
  - “type” emphasizes array status

- Alternative syntax sometimes handy:
  
  ```
  int grade[], i, gpa[];
  ```
  
  - Declares two arrays of base type `int`: `grade`, `gpa`
  - Declares a single `int` variable: `i`

Summary of Arrays

- Arrays are:
  - Sequences of cells holding values of the same type (“base type”)
  - Objects (hence created using `new`)
- To define an array variable:
  ```
  int[] a;  // an array with base type int
  ```
- To create an array object:
  ```
  a = new int[10];
  ```
  - Creates an array of 10 cells
  - The base type is `int`
- To access individual array cells: use indexing
  ```
  a[0], a[1], ..., a[9]
  ```
  - Cells are just like variables:
    - They may be read: `x = a[3];`
    - They may be written: `a[2] = 7;`
A Common Programming Idiom

- To process all elements in array a...
- Do following:
  ```java
  for (int i = 0; i < a.length; i++){
      ...process the one element at a[i]...
  }
  ```
- Use fresh loop counter to avoid overwriting another variable of same name elsewhere
- Remember: Use `i < a.length` not `i <= a.length`

Copying Arrays

- Does the following copy a into b?
  ```java
  int[] a = new int[5];
  int[] b = a;
  ```
  No: a, b are aliases
- How to make a copy?
  ```java
  int[] a = new int[5];
  int[] b = new int[a.length];
  for (int i = 0; i < a.length; i++){
      b[i] = a[i];
  }
  ```
Making Arrays Bigger

- Suppose we want to make an array bigger by adding an element.
- Does the following work?
  ```java
  int[] a = new int[5];
  a.length++;  
  ```
- No!
  - We get the following:
    ```
    Exception in thread "main" java.lang.Error:
    Unresolved compilation problem:
    The final field array.length cannot be assigned
    at Sample.main(Sample.java:15)
    ```
- `a.length` is immutable
- No assignment is allowed

To Make an Array Bigger...

- Create a new larger array object
- Copy old array contents into new object
- Assign address of new object to variable
  ```java
  int[] a = new int[5];
  {
    int[] temp = new int[a.length + 1];
    for (int i = 0; i < a.length; i++){
      temp[i] = a[i];
    }
    a = temp;
  }
  ```
- New variable `temp` created to hold copy
- New block created to ensure `temp` does not interfere with another variable of the same name
- Previous contents of `a` become garbage
Arrays As Arguments

- Arrays = objects
- Array variables = references
- Array cells = variables of the base type (references or primitives depending on what that base type is)
- Both can be used as arguments to methods
  - Array cells: passed just like the variables of that base type
  - Array arguments: passed just like objects
    - Reference to array is passed in
    - If the method expects an array of doubles, an array of doubles of any size can be passed
    - Promotion does not apply. You cannot pass an int array when an array of doubles is expected

Array Initializers

- Arrays may be initialized at declaration time!
  ```java
  int[] a = {5, 0, 1, 2};
  ```
- Java:
  - counts elements (here, 4);
  - creates correct size of array
  - copies elements into array
  - returns reference to array

See Array Example 3
Arrays of Objects

- **Class types** can also be base types of arrays
  - e.g.
    
    ```java
    String[] acc = new String[3];
    ```
  - Array cells store references to objects

- Array initializers can also be used

  ```java
  String[] acc = {"UMD", "UNC", "Duke");
  ```

Arrays of Objects (continued)

- More complicated example than strings:
  - Cat objects

- Expressions can also appear in initializers

  ```java
  Cat[] kennel = {
      new Cat("Joe"),
      new Cat("Jill"),
      new Cat("Fluffy")
  };
  ```
Privacy Leaks

```java
public class MutableThing {
    ...  
    public void mutateMe() {...};
}

public class Foo {
    private MutableThing q = new MutableThing();
    
    public MutableThing getQ() {
        return q;
    }
}
```

- Consider following code
  ```java
  Foo f = new Foo();
  MutableThing m = f.getQ();
  m.mutateMe();
  ```

- After this executes, what happens?
- This phenomenon is called a privacy leak
  - Private instance variables can be modified outside class
  - Behavior is due to aliasing

Fixing Privacy Leaks

- Return copies of objects referenced by instance variables
- To fix `getQ` method in `Foo`:
  ```java
  mutableThing getQ() {
      return new MutableThing(q);  
  }
  ```

- This returns a copy of `q`
- Changes made to this copy will not affect original
Reference Copying

Person[] d = {
    new Person(2.1, 7, ...),
    new Person(3.3, 2, ...)
};

Person[] e = d;

Shallow Copying

Person[] d = {
    new Person(2.1, 7, ...),
    new Person(3.3, 2, ...)
};

Person[] e = new Person[d.length];
for (int i=0; i < d.length, i++){
    e[i] = d[i];
}
Deep Copying

Person[] d = {
    new Person(2.1, 7,...),
    new Person(3.3, 2,...)
};

Person[] e = new Person[d.length];
for (int i=0; i<d.length; i++) {
    e[i] = new Person(d[i]);
}

Three Ways of Copying
CDCollector contains an array of CD’s;
ReCDCollectionOwner p = 
    new ReCD...;
ReCD[] a = p.getCD...();
a[0] = otherCDalreadycreated;
a[0].rewrite(“other”, “name”);

ReCDCollector contains an array of rewritableCD’s;

- **Reference copy**
  ```java
  public ReCD[] getCDsReferenceCopy() {
      return myFavorites;
  }
  ```

- **Shallow copy**
  ```java
  public ReCD[] getCDsShallowCopy() {
      ReCD[] copy = new ReCD[myFavorites.length];
      for (int i = 0; i < copy.length; i++)
          copy[i] = myFavorites[i];
      return copy;
  }
  ```

- **Deep copy**
  ```java
  public ReCD[] getCDsDeepCopy() {
      ReCD[] copy = new ReCD[myFavorites.length];
      for (int i = 0; i < copy.length; i++)
          copy[i] = new ReCD(myFavorites[i]);
      return copy;
  }
  ```
When To Use What Kind of Copying?

- Reference copying is usually a bad idea (not always but realize what you are doing)
- Deep copying provides maximal protection against aliasing (but takes a lot of time and space if it was not necessary)
- Storage space and time used
  - Reference: least
  - Shallow: middle
  - Deep: most
- If the class is mutable, aliasing is something to be avoided and you must have true copies to prevent privacy leaks and modifications outside.
- If you know the class is immutable, aliasing doesn’t hurt but neither does making true copies (except wasted space and time).
- If storage is an issue, aliasing problems may be worth coping with but must be well documented.